A MACHINE-LEARNING METHOD TO EXPLORE THE UEFI LANDSCAPE

How we found unwanted UEFI components hidden in a sea of millions of samples

Authors:
Filip Mazán
Frédéric Vachon

CONTENTS

INTRODUCTION . . . 2

BASICS OF UEFI . . . 2

- High-level overview of a UEFI-compliant firmware boot flow . . . 2
- UEFI firmware layout . . . 3
- UEFI executable file format . . . 3

BUILDING A PROCESSING PIPELINE FOR UEFI EXECUTABLES . . . 4

- Motivations . . . 4
- Analyzing UEFI executables in the context of machine learning . . . 5
- Feature engineering. . . . 6
  - Strings representation . . . 6
  - Multidimensional sequential data. . . . 7
  - Product of the feature engineering . . . 9
- Data representation and processing pipeline . . . 9
  - Nearest neighbors . . . 10
  - Similarity score . . . 11
- Lessons learned designing our processing pipeline . . . 12

DISCOVERIES . . . 13

- UEFI firmware backdoors . . . 13
  - What is a UEFI firmware backdoor? . . . 13
  - Prevalence of UEFI firmware backdoors . . . 13
  - Case study: ASUS backdoor . . . 14
  - Case study: HP Backdoor . . . 15
- OS-level persistence modules . . . 17
  - Overview of known software with firmware persistence. . . . 17
  - Observed techniques . . . 18
  - Case study: Lenovo Service Engine . . . 18
  - Case study: Samsung's SecureGuard . . . 22
  - Case study: HP Sure Run . . . 25
- Security implications . . . 26

CONCLUSION . . . 27

Authors: Filip Mazán and Frédéric Vachon

September 2019

INTRODUCTION

UEFI (Unified Extensible Firmware Interface) security has been a hot topic for the last few years. Several high-impact vulnerabilities have been found, and even a few *rootkits* exposed. Finding such rootkits in the wild is a very challenging task, comparable to finding a needle in a haystack. In this paper, we share the results of our year-long research looking for unsafe UEFI components based on a data pool of over two million unique samples. We describe the techniques that we used to filter the sample set down to the most promising candidates for human analysis. We also provide a comprehensive analysis of the most interesting unsafe components we identified.

In the first part of the paper, dedicated to the outlier identification, we describe the process of extracting UEFI executables' features from the results of both static and behavioral analysis. Then, we detail the inner workings of the custom, real-time pipeline used to cluster samples and put outliers under the spotlight. This has led to the creation of an internal tool that can be used by analysts to leverage the pipeline output and concentrate their efforts towards the most relevant samples to investigate.

The second part of the paper focuses on the analysis of the most interesting outliers, which we classified in two distinct categories: UEFI firmware backdoors, and persistence components for OS-level software. We provide technical details about the various techniques used by the components to achieve their goal, such as using the Windows Platform Binary Table (WPBT) mechanism to drop and run executables, or replacing Windows binaries by writing directly to the file system.

BASICS OF UEFI

Since a significant part of this paper is dedicated to processing UEFI executables, we begin with a brief introduction to UEFI firmware. UEFI is a *specification* defining the interface that exists between the OS and the device's firmware. There is an open source reference implementation called EDK II that is maintained by TianoCore and hosted on *GitHub*. The specification defines a set of standardized services, called "boot services" and "runtime services", that are the core APIs available in UEFI firmware. Additionally, firmware functionalities can be extended by registering new services called "protocols" that can be retrieved by other firmware components. Bootloaders can then use these standardized interfaces to boot the OS.

High-level overview of a UEFI-compliant firmware boot flow

A device with UEFI-compliant firmware will go through multiple phases when it boots. It will first go through the SEC (security) phase which is the root of trust of the boot chain. Then, the boot goes through the PEI (Pre-EFI Initialization) phase where core hardware initialization occurs. After this, the system transitions to the DXE (Driver eXecution Environment) phase. During this phase, the DXE dispatcher loads all the DXE drivers it can find in the firmware volumes. That's when the boot services and runtime services are populated and additional protocols are created. Finally, the volume where the bootloader is located is chosen and the OS loader is executed. The UEFI executables that we processed in the course of this research are those that run from the DXE phase onwards.

UEFI firmware layout

The UEFI firmware is stored in SPI flash memory, which is a chip soldered on the system's motherboard. Thus, replacing one of the drives or formatting the system doesn't affect the firmware code.

UEFI firmware is very modular: it usually contains dozens, if not hundreds, of executables. To store all these separates files, the firmware is laid out in volumes using the Firmware File System (FFS), a file system specifically designed to store firmware images. The volumes contain files that are identified by GUIDs and each of these files contain one or more sections holding the data. One of these sections contains the actual executable image. Figure 1 shows an example UEFI FFS listing. We used *UEFITool*, an open source project for manipulating UEFI firmware images, to help visualize this.

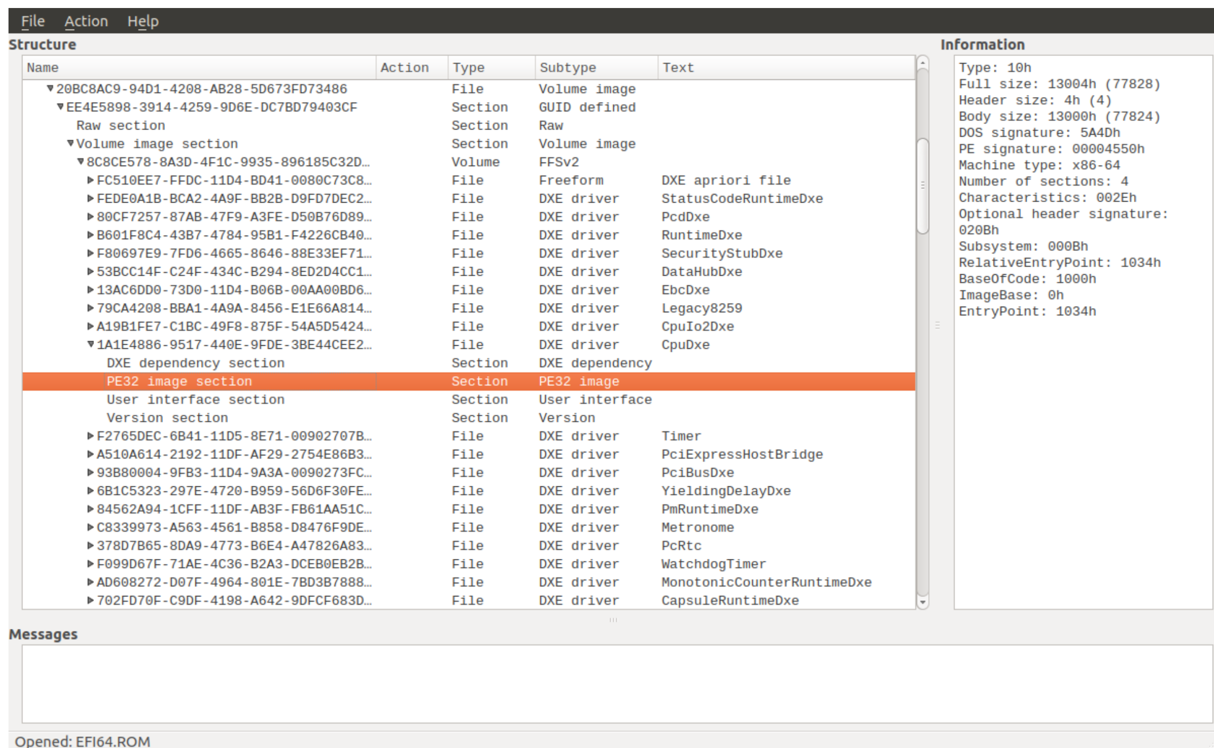

Figure 1 // Example UEFI firmware as displayed in UEFITool

UEFI executable file format

There are two executable file formats that are supported by UEFI firmware. The first one is Microsoft's Portable Executable (PE) format. That's the most common one, especially when it comes to DXE drivers and OS loaders. This is very beneficial for binary analysis since this means that the tooling built upon Windows executables will most likely work with UEFI executables as well. The Subsystem field of the structure IMAGE_OPTIONAL_HEADERS tells if the PE is a UEFI executable. There are four values that are used by UEFI executables:

- `IMAGE_SUBSYSTEM_EFI_APPLICATION`
- `IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER`
- `IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER`
- `IMAGE_SUBSYSTEM_EFI_ROM`

The second format is called the Terse Executable (TE) format; it is used by Pre-EFI Initialization Modules (PEIM) that are loaded during the Pre-EFI Initialization phase. A lot of the PE header fields are unused by UEFI executables. For instance, most of the Data Directories are never used, Windows-specific fields such as MajorOperatingSystemVersion and MinorOperatingSystemVersion are set to 0, the MZ header is useless and so on. Because of this and the limited space available to store firmware images, a new executable file format was included in the specification. It is an aggressively stripped version of the PE format. The MZ header, the COFF header and the optional headers are replaced by the single structure displayed in Figure 2, thus giving a very compact file format specifically tailored for the UEFI environment.

Prototype

```
typedef struct {
  UINT16                     Signature;
  UINT16                     Machine;
  UINT8                      NumberOfSections;
  UINT8                      Subsystem;
  UINT16                     StrippedSize;
  UINT32                     AddressOfEntryPoint;
  UINT32                     BaseOfCode;
  UINT64                     ImageBase;
  EFI_IMAGE_DATA_DIRECTORY   DataDirectory[2];
} EFI_TE_IMAGE_HEADER;
```

Figure 2 // Definition of the TE image header from the PI specification 1.6

BUILDING A PROCESSING PIPELINE FOR UEFI EXECUTABLES

In this part of the paper, we cover the specifics of the processing pipeline that we built to drastically reduce the number of samples that require human attention. We first explain the motivations behind this project. Then, we go through the different choices that we made regarding the feature extraction process. Finally, we cover our experimentation with various algorithms to compute a similarity score between UEFI executables and hence to identify outliers.

Motivations

Very little UEFI-based malware has been found in the past. One of the reasons for that is that, until recently, no security products had visibility into the firmware. Thanks to the telemetry gathered by ESET's UEFI scanner, we were in a favorable position to start hunting for such malware.

The first problem we faced when looking at this task was the large amount of data we needed to dig through. To give an idea of the scale, in the past two years, we've seen over 5.5 billion UEFI executables out of which 2.5 million are unique. These files were not only collected from our customers, but also from public sources such as trusted vendor updates, GitHub repositories and so on. Since we had limited staff resources available to process these samples, it quickly became obvious that we needed some automated system to reduce this to something that could realistically be processed by a few analysts.

Hence we decided to build a system tailored to highlight outlier samples by finding unusual characteristics in UEFI executables. By building this system, we were able to reduce the analysts' workload by as much as 90% had the analysts been required to analyze all the new samples we receive. Let's now look at our journey creating this system by demonstrating some of the aspects we investigated.

Analyzing UEFI executables in the context of machine learning

Although in recent times artificial intelligence (AI) and, more concretely (ML) are very widely and commonly used terms and are often presented as a silver bullet, they are under no circumstances a solution for every kind of problem and every kind of input data. It is the data scientist's job first to assess the situation, next to study the domain, and then work iteratively and in small steps in order to finally achieve a desired outcome.

The first step in every ML-based approach is to understand the data we are working with – in our case, UEFI executables – and try to find a way to transform the data into a form that machine learning algorithms can easily understand and work with. This meant that we had to devise a process that takes UEFI executable as inputs and that outputs so-called features. In ML jargon, we can understand features as individually measurable properties or characteristics of observed data, usually in a numeric format. Choosing the correct and descriptive features are crucial factors to the success of every ML-based solution.

In the context of UEFI executable files we have identified two distinct ways how to look at them:

- static analysis – looking at the executable as is, its structure (PE header, sections, procedures, strings), disassembly, calculated statistics (entropies, occurrences, ratios)
- behavioral analysis – how the sample behaves when executed in an emulation engine, inspecting code and data flows, gathering statistics about the execution

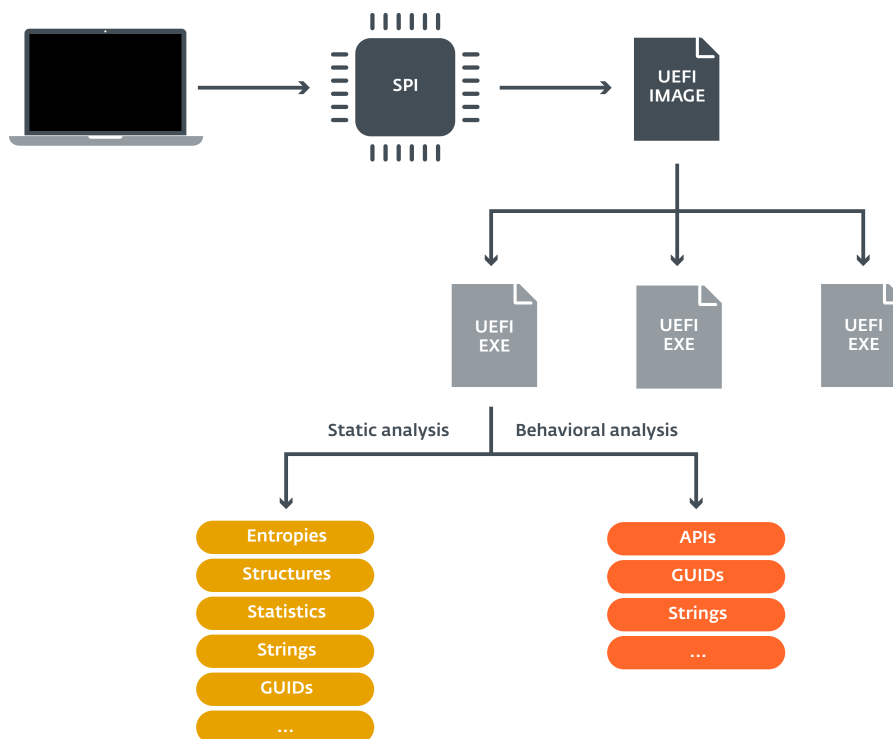

Figure 3 // The process behind extraction of UEFI executables from SPI flash and extraction of artifacts

By extracting information about the executable files using both static and behavioral analysis, we can gather thousands of distinct artifacts associated with every single executable file. Some of these artifacts are already numeric and can be easily treated as features (e.g. file size, some of the PE header entries, number of procedures, strings count), but there are many artifacts that are not in a numeric format and need to be converted.

Feature engineering

As was previously mentioned, not all artifacts extracted from both static and behavioral analysis of executable UEFI files can be simply converted to numeric format, which most of the learning algorithms require.

As an example of this kind of artifact transformation we demonstrate the process behind transforming retrieved strings from the binaries and embedding multidimensional sequential data into numeric vectors.

Strings representation

Strings transformation and representation is a well-known topic in natural language processing and many different techniques have been devised for distinct applications. The most commonly used techniques for this task, ordered by increasing complexity, are:

- *term frequency* vectorizer
- *tf-idf – (term frequency – inverse document frequency (tf-idf)* statistic
- *latent semantic analysis (LSA)*[1] described and patented in 1988 based on *singular value decomposition (SVD)*
- *word2vec*[2], *doc2vec*[3]

For the purpose of transforming strings found by the analysis of UEFI executable files, we have chosen the third technique – latent semantic analysis. This approach was chosen mainly due to the ease of use of the LSA transformation and its satisfactory performance on our dataset. In the future we plan to experiment also with the other techniques and compare their impact on the final outlier detection solution.

At first, all the strings retrieved from all the binaries are transformed to n-grams (n=3), then term frequency and tf-idf statistics are computed and finally a truncated SVD is performed on the whole training dataset. From our testing results we concluded that the tf-idf statistic was less performant in terms of cumulative explained variance than pure term frequency. This comparison can be seen in the figure below.

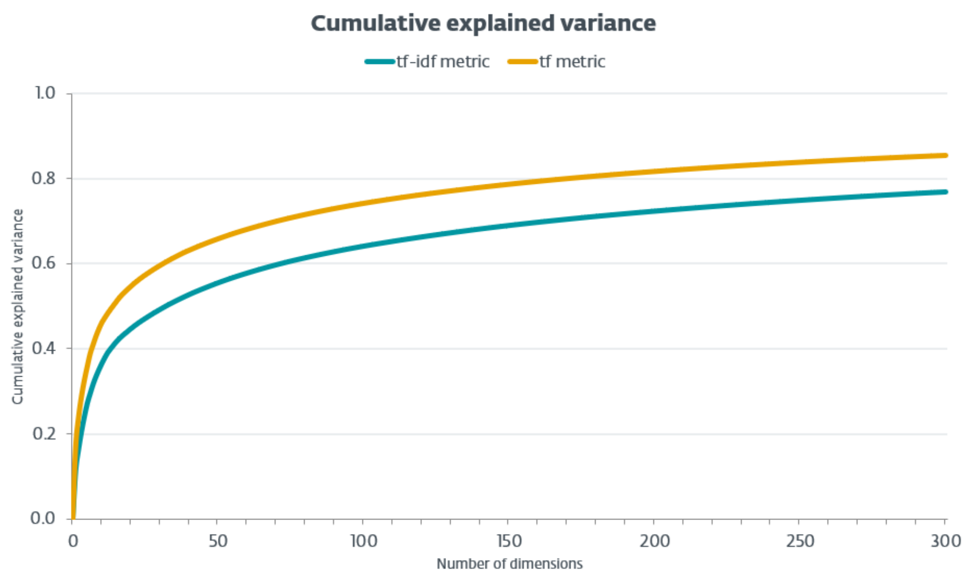

Figure 4 // Cumulative explained variance plotted against increasing number of embedding dimensions

1 DEERWESTER, Scott, et al. Indexing by latent semantic analysis. Journal of the American society for information science, 1990, 41.6: 391-407.
2 MIKOLOV, Tomas, et al. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013.
3 LE, Quoc; MIKOLOV, Tomas. Distributed representations of sentences and documents. In: International conference on machine learning. 2014. p. 1188-1196.

The process of transformation of every new binary sample begins with string extraction, n-gram creation, term frequency calculation and application of truncated an n-dimensional vector that can fairly accurately describe the content of all the retrieved strings. The cumulative explained variance ratio, depending on the length of the resulting vector, is shown in Figure 4. Based on the tradeoff between the space requirements for storing the vectors and explained variance, we have chosen an appropriate dimension count.

Multidimensional sequential data

During the analysis of UEFI executables, many sequential artifacts can be extracted – that means it's crucial to utilize not just their value, but also the order in which they appear. As an example, we are showing 2 cases – one with three selected statistics about PE sections in the UEFI binary and the other with a 66x4 other type of statistics extracted from the binary.

It is widely known that PE-formatted executable files, which most UEFI binaries are, are composed of one or more sections. These sections represent logical and physical units of either code or data and various statistics can be computed from them. To name two statistics that are used in our approach, we utilize section size and section entropy.

Naturally, with an increasing number of statistics and observations for each PE section, the total number of dimensions rises. It is unwise to use the whole multidimensional vector in the final outlier computations due to y reduction.

Since our data in this case are multidimensional and sequential, we first experimented with *convolutional* n-dimensional *auto-encoding neural networks*. The goal of this kind of auto-encoder is to apply multiple kernels to reduce the dimensionality of the input, while trying to minimize the reconstruction error (this means to try to reconstruct the original data point from the latent space while keeping as much of the original information as possible).

The training of the convolutional auto-encoder with *Adam optimizer*, loss defined by mean squared error and validation split of 20% converged very quickly, and the resulting reconstruction loss was minimal. Figure 5 shows a random example from the 66x4 sized statistics being reduced to two different latent space sizes – 8 and 32 – and the model's reconstruction attempt from it. Even by visual inspection, it can be clearly seen that the auto-encoder can successfully reduce dimensionality to a small fraction of the original data, which considerably increases the speed and lowers the memory usage of the whole outlier detection solution.

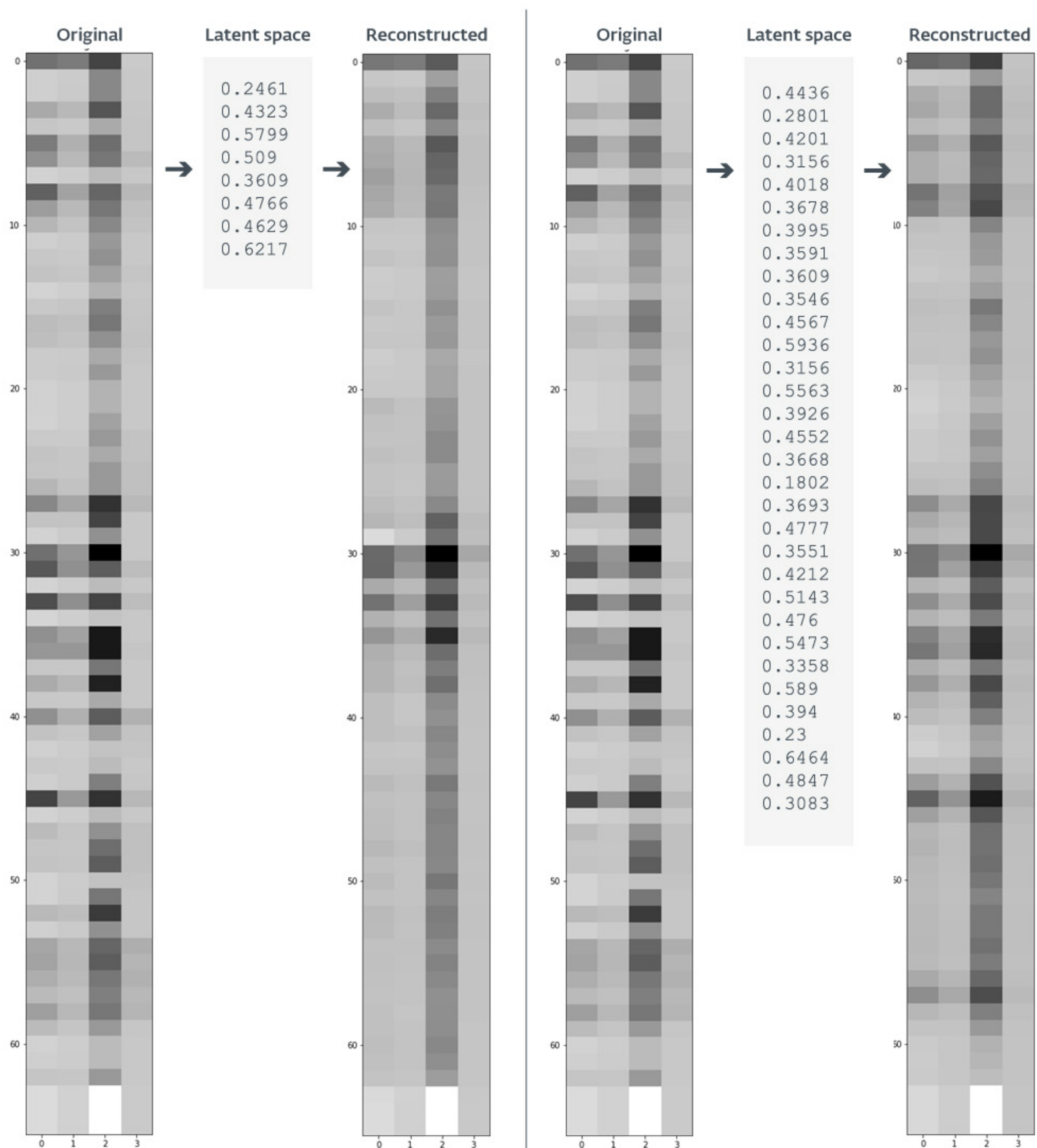

Figure 5 // Example of embedding original sequential data into 8 (left) and 32 (right) latent dimensions with the corresponding data reconstruction. In layman's terms, the original data (visualized by grayscale image on the left) is transformed into a fixed-length latent space (represented by numbers), while keeping as much of the original information in it. This is demonstrated by reconstructing the original data from this latent space (visualized by grayscale image on the right) which should very closely resemble the original data. As we can see, the reconstruction from a 32-dimensional latent space was able to better capture even minor details compared to just 8-dimensional space.

We have also conducted a comparison of multidimensional convolutional auto-encoder with a simpler multi-layered perceptron (MLP) auto-encoder. With lower latent space size, the convolutional model performed better than the MLP model. With growing latent space, both models' reconstruction loss converged. In the end, we decided to use the convolutional auto-encoder in order to reduce original data dimensionality more than the MLP would be able to with the same latent space size.

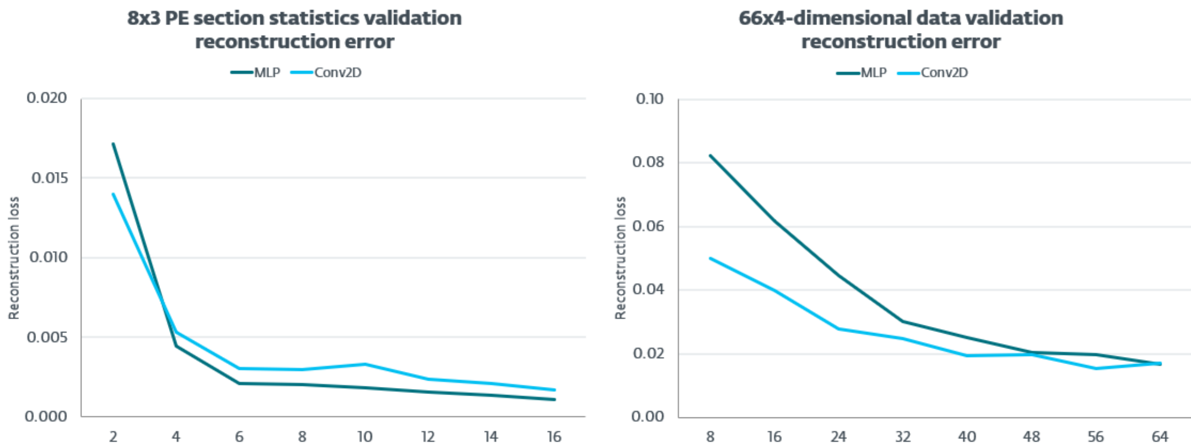

Figure 6 // Reconstruction losses for smaller dataset dimensionality (left) and a bigger one (right) for two distinct auto-encoder architectures – convolutional (Conv2D) and multi-layered perceptron (MLP)

Product of the feature engineering

To sum it up, after the feature engineering phase, every executable UEFI binary can be transformed into a multidimensional vector by using various preprocessing steps, such as string transformations, *auto-encoders*, *binnings*, *one-hot encodings* and other techniques.

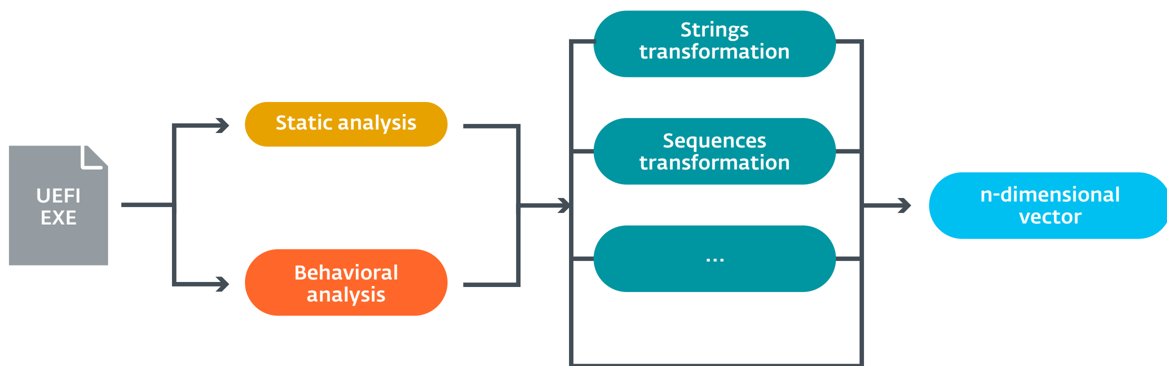

Figure 7 // High level overview behind the transformation of a UEFI executable into a vector space

Data representation and processing pipeline

Our defined goal is to find oddities among incoming UEFI binaries. That means we have to define a metric by which we can tell how similar or close any two UEFI binaries are in their vector spaces, in order to examine whether any incoming UEFI binary is similar to what we have already seen and how similar it is. Many metrics were evaluated, like *cosine distance*, *Manhattan distance* and *Euclidean distance*. The one that performed the best and that is compatible with the data structure we are using for storage and indexing was plain Euclidean distance.

As we previously outlined, the key component of determining whether a UEFI binary is an oddity is to find the binaries most similar to it and analyze their distances and the sample neighborhood as a whole in Euclidian space.

Nearest neighbors

There exist many algorithms for finding the nearest neighbors in Euclidean space, some of which we mention in this next section.

The *naïve brute-force approach* relies on sequential distance calculations for each sample in the dataset. The performance of brute-forcing is, however, unsatisfactory, as it scales as $O(DN^2)$, where D represents number of dimensions and N the number of samples in the dataset. In early testing on our dataset, calculating the nearest neighborhood for one UEFI sample took over 2.5 seconds. This is a very high number, considering we are planning to use these neighborhood searches in real time and we will be making possibly millions of such calculations in our ever-growing dataset.

Instinctively we knew we had to look for other solutions not based on brute-force – ideally, tree-based – as trees are well known data structures with beneficial performance characteristics. At first, we tried a simple *k-dimensional tree*, which can efficiently remove large parts of the search space while searching. The downside of k-d trees is their inability to handle higher dimensionality data efficiently. Generally speaking, the number of samples, N, should be much larger than 2^k, for k-dimensional space, or $N >> 2^k$. If this condition is not met, their performance degrades drastically, beyond the point of efficiency of brute-force approach. Another downside of trees in general is that they take some time to construct, which in our case was 3 minutes and 26 seconds. Evaluation of k-d trees on our dataset yielded an average time needed for calculating nearest neighborhood of 0.027 seconds, which is already significantly less that the brute-force approach.

Another tree-based approach that should not suffer from performance degradation in higher dimensions is a *ball tree*. This tree gets its name from the fact that it organizes its nodes as high dimensional hyperspheres known as balls defined by their centroid and radius. Evaluation of ball trees on our dataset yielded an average time needed for calculating nearest neighborhood of 0.17 seconds and the construction time of over 8 minutes.

The last tree-based approach we evaluated for our nearest neighbor searching effort is a *vantage-point tree* (vp-tree). By definition, vp-trees are similar to ball trees, as they divide space by using hyperspheres. The difference is that vp-trees partition the space using only one hypersphere per node and the data points either lie in the hypersphere or out of it, whereas the ball trees partition the data points strictly inside the defined hyperspheres, which may overlap. Evaluation of vp-trees on our dataset yielded an average time needed for calculating nearest neighborhood of 0.023 seconds and the construction time of approximately 2 minutes and 54 seconds.

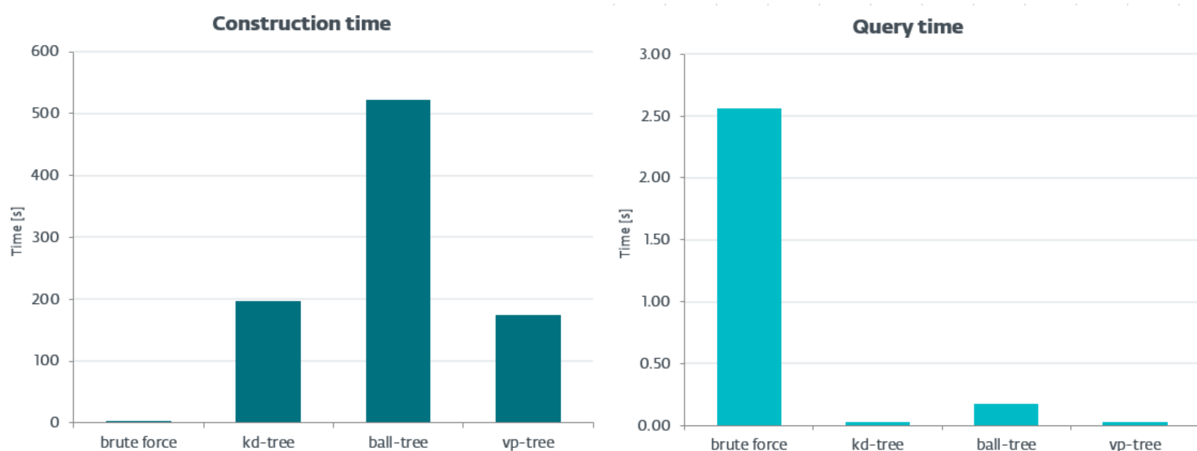

Figure 8 // Comparison of construction and query time of four nearest neighborhood finding approaches on our dataset

According to the results, the native vp-tree library performed the best in regards both to the construction time and query time, and that is why we have chosen to use it in the real-time process of searching nearest neighbors.

Similarity score

Now that we can quickly and efficiently retrieve the nearest neighborhood to any UEFI executable, we can analyze all the relevant neighbors and their relation to the target executable.

One of the factors we use to compute the final verdict about an incoming UEFI executable is the layout of its local neighborhood. At first, we analyze information about possible local clusters within the region. To find out whether there are sample clusters present, we run *DBSCAN*[4]. An example visualization of a *T-SNE projection*[5] of a sample neighborhood with clusters marked is presented in the Figure 8. Next, the distances to nearest files are processed, and the system then evaluates whether the target executable is likely to be known benign, known malicious, or an oddity that should be inspected by an analyst. The result of this process is, that each incoming executable is assigned a “similarity” score in the range of zero to one. Files with the lowest similarity score are then inspected by an analyst with the highest priority.

Figure 9 // Example of clusters found in samples' local neighborhood

Finally, each new incoming UEFI executable file is added back to all the data structures and dataset, so the whole process can be running in real-time and the results can be up-to-date with each new incoming file. This achieves the correctness of the process in circumstances, when e.g. new firmware is released and there is a burst of never-seen-before files, out of which the analysts need to inspect only the representative samples.

4 ESTER, Martin, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Kdd. 1996. p. 226-231.
5 MAATEN, Laurens van der; HINTON, Geoffrey. Visualizing data using t-SNE. In: Journal of machine learning research, 2008. p. 2579-2605.

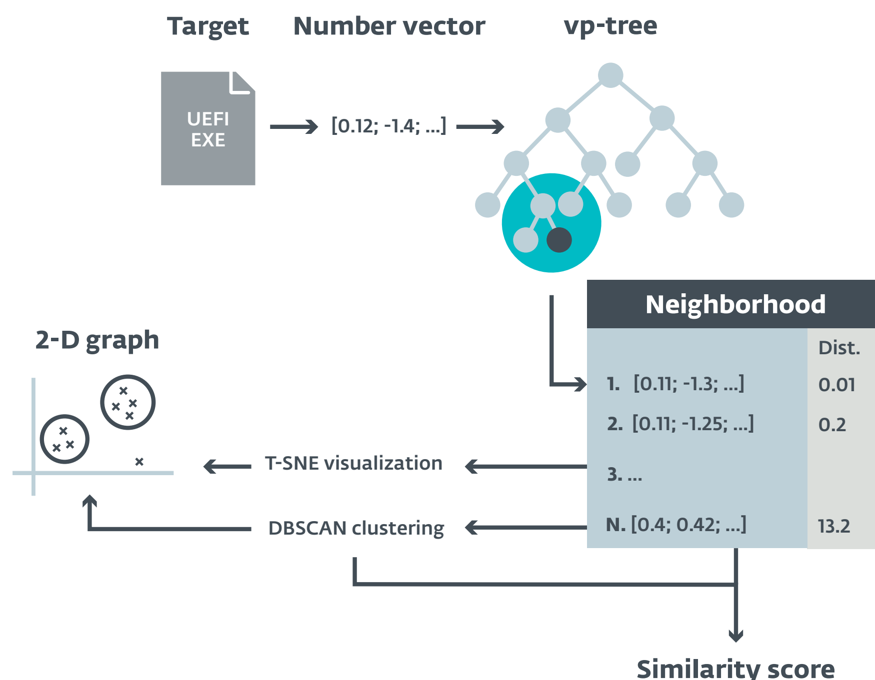

Figure 10 // The pipeline behind each new incoming UEFI executable

As a proof of concept, we tried to evaluate the whole system on known suspicious and malicious UEFI executables, most notably the LoJax DXE driver, which had not previously been included in our dataset. The system successfully concluded that the LoJax executable was very dissimilar to anything we had ever seen before and assigned it a similarity score of 0. The closest UEFI executables to this LoJax executable were found to be NTFS drivers and other applications utilizing NTFS, but the distances to them were still fairly large. This is an expected result, LoJax executable embeds an NTFS driver to be able to drop malicious content. All in all, this test gives us a degree of confidence that if another similar UEFI threat emerged, we would be able to identify it as an oddity, so the analysts could investigate it with high priority.

Lessons learned designing our processing pipeline

In this part of the paper, we have presented ESET's ML-based approach for discovering oddities in the vast landscape of UEFI executables. Given that we have seen over 2.5 million unique UEFI binaries in the past two years alone, the need for automating the process arose naturally. In our research efforts we have been examining and comparing multiple approaches of every part of the process, from feature extraction, text embedding, embedding multidimensional data through efficient storage and querying of samples' neighborhoods to generate a final scoring algorithm. All of this was done while taking into account performance and real-time capabilities of the techniques chosen.

With an efficient method of retrieving the nearest neighbors to any incoming UEFI binary, we have set up a system of assigning similarity scores to these incoming executables, comparing them to previously seen files. Using the LoJax DXE driver – a known malicious file that was not included in our dataset – as a proof of concept, we demonstrated the system's strong capabilities in identifying oddities in the sea of UEFI samples. Our findings make us confident that should a new UEFI threat emerge, we will be able to spot and promptly analyze it.

According to the statistics collected from the system, the ML-based approach can reduce the workload of our analysts by up to 90% (if they were to analyze every incoming sample). Further, thanks to the fact that each new incoming UEFI executable file is added back to the data structures and dataset, our solution offers real-time monitoring of UEFI landscape.

DISCOVERIES

In the second part of this paper, we present the results of our hunting using the system described above. The interesting components that we found can be grouped in two categories: UEFI firmware backdoors and persistence for OS-level software. For each of these categories, we provide an explanation of what they are and we analyze the cases we deemed the most interesting. Finally, we discuss the security issues that are introduced by embedding such components in UEFI firmware.

UEFI firmware backdoors

What is a UEFI firmware backdoor?

The first group of firmware components that we cover here are UEFI firmware backdoors. Let's first describe what they are. In most of the UEFI firmware setups, options are available to password protect the system from unauthorized access during the early stages of the boot process. The most common options allow setting passwords to protect access to the UEFI firmware setup, to prevent the system from booting and to access the disk. UEFI firmware backdoors are mechanisms that allow bypassing these protections without knowing the user-configured password. They are sometimes triggered by pressing a specific combination of keys when asked to enter the password and sometimes activated when a user enters multiple invalid passwords in a row. For the latter, the UEFI firmware setup is usually locked by displaying a code from which the backdoor password can be computed. Such a dialog is shown in Figure 11.

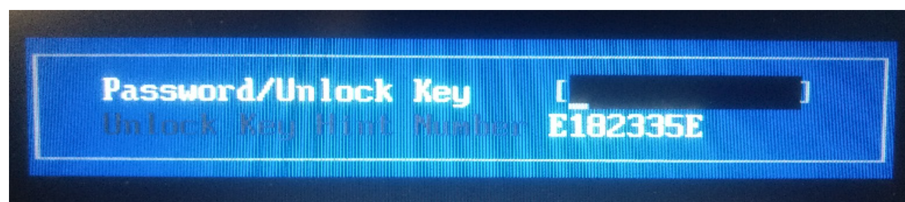

Figure 11 // Example of a backdoor access prompt

While it could be argued that these mechanisms would rather be called recovery mechanisms than backdoors, we're using the latter because that is how they were named by the firmware developers themselves. In a lot of the cases, the mechanism either used UEFI variables or filenames containing the word “Backdoor”.

Prevalence of UEFI firmware backdoors

UEFI firmware backdoors are very common. They are used as a recovery mechanism in case the computer's owner forgets the password. Password generators for multiple OEMs firmware are available online. As an example, this *blogpost* contains generators for many vendors, such as HP, ASUS, Sony and Samsung. Most importantly, it also provides generators for Insyde Software and Phoenix Technologies' firmware. Insyde Software, Phoenix Technologies and American Megatrends Incorporated are the three companies developing base implementations of UEFI firmware that they sell to OEMs who can then customize them to their needs. Having backdoors implemented at this level means that they are likely to be included in all the firmware of OEMs buying their base implementation from Insyde Software and Phoenix Technologies.

We present two UEFI firmware backdoors that we analyzed during this research to illustrate the two types of backdoor we mentioned. We'll cover a backdoor present in ASUS laptops and then another one that we found on some HP notebooks.

Case study: ASUS backdoor

Among all the backdoors we analyzed, the ASUS backdoor is by far the most prevalent. The name doesn't come from us; it was used in some ASUS firmware where the executable responsible for validating the backdoor password was called `AsusBackDoor`. We found this backdoor in all the ASUS laptops we examined. Here's the list of models we know are affected, but the list should be longer:

- Taichi 31
- ZenBook UX301LA
- ROG G550JK
- ROG ZEPHYRUS (GX501)
- ZenBook UX430UQ
- VivoBook Max X541UA

The ASUS UEFI firmware setup usually has three different type of protections that can be configured:

- Administrator password: Password prompt to access the UEFI firmware setup
- User password: Password prompt upon system boot
- HDD Password: Access to the disk is password protected

When prompted for one of these passwords, pressing ALT+R will open the backdoor prompt. Depending on the laptop models, entering the right password will either remove all of these passwords or only the administrator and user password.

In most of the cases we looked at, the UEFI executable responsible for validating the password is a DXE driver that installs a protocol identified by the `GUID c3940226-bf56-4f4f-941b-dba05b7ec3c1`. This protocol is then resolved by *AMITSE*, the UEFI executable implementing the UEFI firmware setup, also called the "Text Setup Environment".

Figure 12 shows one implementation of the code executed when the backdoor prompt validates a password. This code is from the UEFI firmware version 300 of an ASUS ROG GL552VW. At first, the backdoor protocol is resolved using `LocateProtocol`, then the user input is validated using the API exposed via this protocol. If the validation succeeds, the first 0x50 bytes plus two additional fields of the `AMITSESetup` variable are zeroed out, thus disabling the user and administrator UEFI firmware passwords.

```
if ( ReadInput(Input, 0x14ui64, v99, v98, 0i64, 1) )
  break;
AsusBackdoorProtocolGuid.Data2 = 0xBF56u;
AsusBackdoorProtocolGuid.Data3 = 0x4F4F;
AsusBackdoorProtocolGuid.Data1 = 0xC3940226;
AsusBackdoorProtocolGuid.Data4[0] = 0x94u;
AsusBackdoorProtocolGuid.Data4[1] = 0x1B;
AsusBackdoorProtocolGuid.Data4[2] = 0xDBu;
AsusBackdoorProtocolGuid.Data4[3] = 0xA0u;
AsusBackdoorProtocolGuid.Data4[4] = 0x5B;
AsusBackdoorProtocolGuid.Data4[5] = 0x7E;
AsusBackdoorProtocolGuid.Data4[6] = 0xC3u;
AsusBackdoorProtocolGuid.Data4[7] = 0xC1u;
AsusBackdoorProtocol = 0i64;
if ( (gBootServices->LocateProtocol(&AsusBackdoorProtocolGuid, 0i64, &AsusBackdoorProtocol) & 0x8000000000000000ui64) == 0i64 )
{
  for ( i = 0i64; i < 0x14; ++i )
  {
    v25 = Input[i];
    if ( v25 > 0x60u )
      Input[i] = v25 - 0x20;
  }
  if ( !(AsusBackdoorProtocol->ValidatePassword)(Input, 8i64) )
  {
    if ( *qword_46270 <= 5u || (AMITSESetup = GetVar(5u, &VarSize)) == 0i64 || VarSize != 83 )
      AMITSESetup = LoadAndGetVar(5i64, &VarSize);
    MemSet(AMITSESetup, 0x28ui64, 0);
    // Hex-Rays fails at displaying the BYTE* correctly (0xA * 4 = 0x28)
    // Read &AMITSESetup[0x28]
    MemSet(AMITSESetup + 0xA, 0x28ui64, 0);
    AMITSESetup[0x52] = 0;
    AMITSESetup[0x51] = 0;
    if ( *qword_46270 > 5u )
    {
      VariableEntry = (qword_46270 + *(qword_46270 + 6));
      if ( VariableEntry )
        gRuntimeServices->SetVariable(
          VariableEntry->VariableName,
          &VariableEntry->VendorGuid,
          VariableEntry->Attributes,
          0x53ui64,
          AMITSESetup);
    }
```

Figure 12 // Hex-Rays output of the password clearing code in AMITSE

We notified ASUS about this backdoor in April 2019 and they released *firmware updates* on 14. June 2019. We manually looked at the updates for models ROG G550JK, ZenBook UX301LA and VivoBook Max X541UA and can confirm that the backdoor we described here was removed. If you own any ASUS notebook, we strongly advise you to apply the latest firmware update from the *ASUS website*.

Case study: HP Backdoor

The other example of a UEFI firmware backdoor that we cover here was found on HP Pavilion 15 and HP Spectre x360 firmware. It's very likely there are other HP notebooks affected as well. This backdoor is of the second type we mentioned earlier. It is activated when a user enters the wrong UEFI firmware password too many times.

When this happens, the UEFI firmware backdoor is activated and access to the UEFI firmware setup is blocked and a dialog displays a 32-bit value in its hexadecimal form as shown in Figure 13, which contains a picture taken by a user asking for help on the HP Community forum.

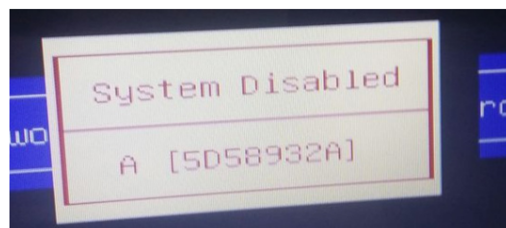

Figure 13 // Example of an HP UEFI firmware setup locked after too many failed password attempts

The user is then asked to enter the correct backdoor password derived from this value.

Under the hood, when too many incorrect passwords are entered, the system generates a value using the rdtsc instruction and stores it in a UEFI variable called `BackDoor`. Figure 14 shows the routine implementing this.

```
VendorGuid.Data2 = 0x93CAu;
VendorGuid.Data1 = 0x8BE4DF61;
VendorGuid.Data4[0] = 0xAAu;
VendorGuid.Data4[1] = 0xD;
VendorGuid.Data4[2] = 0;
VendorGuid.Data3 = 0x11D2;
VendorGuid.Data4[3] = 0xE0u;
VendorGuid.Data4[4] = 0x98u;
VendorGuid.Data4[5] = 3;
VendorGuid.Data4[6] = 0x2B;
VendorGuid.Data4[7] = 0x8Cu;
NbIter = 4i64;
do
{
  TimeStampCounter = rtdsc() >> 3;
  BackdoorData = TimeStampCounter + (BackdoorData << 8);
  gBS->Stall(TimeStampCounter);
  --NbIter;
}
while ( NbIter );
gRT->SetVariable(L"BackDoor", &VendorGuid, 7u, 4ui64, &BackdoorData);
```

Figure 14 // Hex-Rays output of the routine generating the backdoor password value

The generated value in the `Backdoor` variable is retrieved when the user submits a backdoor password. The user input, which is the hexadecimal representation of a 32-bit value, is then converted to an integer. A checksum is computed based on a hardcoded key and the "`Backdoor`" value. If the result matches the user input, the system is unlocked. The code responsible for this is shown in Figure 15.

```
Key[0] = 0xB9u;
Key[1] = 0xEDu;
Key[2] = 0xF5u;
Key[3] = 0x69;
Key[4] = 0x9Du;
Key[5] = 0x16;
Key[6] = 0x49;
Key[7] = 0xF9u;
Key[8] = 0x8Cu;
Key[9] = 0x5F;
Key[10] = 0x7C;
Key[11] = 0xB3u;
Key[12] = 0x68;
Key[13] = 0x3C;
Key[14] = 0xD4u;
Key[15] = 0xA7u;
if ( (gRT->GetVariable(L"BackDoor", &VendorGuid, 0i64, &Size, &BackDoor) & 0x8000000000000000ui64) == 0i64 )
{
  StrToDword(&InputDword, UserInput, 0);
  BackDoorChecksum = CalcChecksum(Key, &BackDoor);
  if ( InputDword == BackDoorChecksum )
  {
    v16 = 3i64;
    ResetAMITSE();
  }
}
```

Figure 15 // Hex-Rays output of the routine validating the backdoor password

We have seen other similar, UEFI firmware backdoors during our research. The behavior being very similar to the ones we described, we have limited ourselves to describing only these two. Security issues related to these firmware components will be covered in the Security implications section. The following part of the paper is dedicated to firmware components used as persistence mechanisms for Windows software.

OS-level persistence modules

During the course of our research, we have uncovered multiple firmware components that were responsible for installing software at the operating system level. In this section, we give an overview of the publicly known software that is installed from the UEFI firmware. We then describe the techniques that are used to achieve this goal. Finally, we provide a technical analysis for the cases that stand out.

Overview of known software with firmware persistence

Probably the most well-known software to have a persistence mechanism in the firmware is Absolute Software's LoJack, previously known as Computrace. LoJack is anti-theft software that allows tracking and helping to recover stolen devices.

Since thieves are likely to format the disk of the stolen device: to be effective, the anti-theft solution has to survive this kind of manipulation. To achieve this goal, the persistence module was added to the firmware. Anti-theft software is the reference use-case for installing OS-level software from the firmware.

However, this technique was quickly adopted by some OEMs to provide additional persistence to their preloaded software. In 2015, it became known publicly that Lenovo had added a module to its firmware to install Lenovo Service Engine (LSE). According to the Chinese company, this program was used to send non-identifiable system information to their server. It was also responsible for installing Lenovo OneKey Optimizer, software they describe as "a powerful, next-generation system optimization software designed specifically for Lenovo computers [that] can enhance your PC's performance by updating the firmware, drivers, and pre-installed apps [and] also provides power management schemes that can extend the life of your battery."[6] LSE was quickly removed from the firmware after researchers found that it was affected by vulnerabilities.[7]

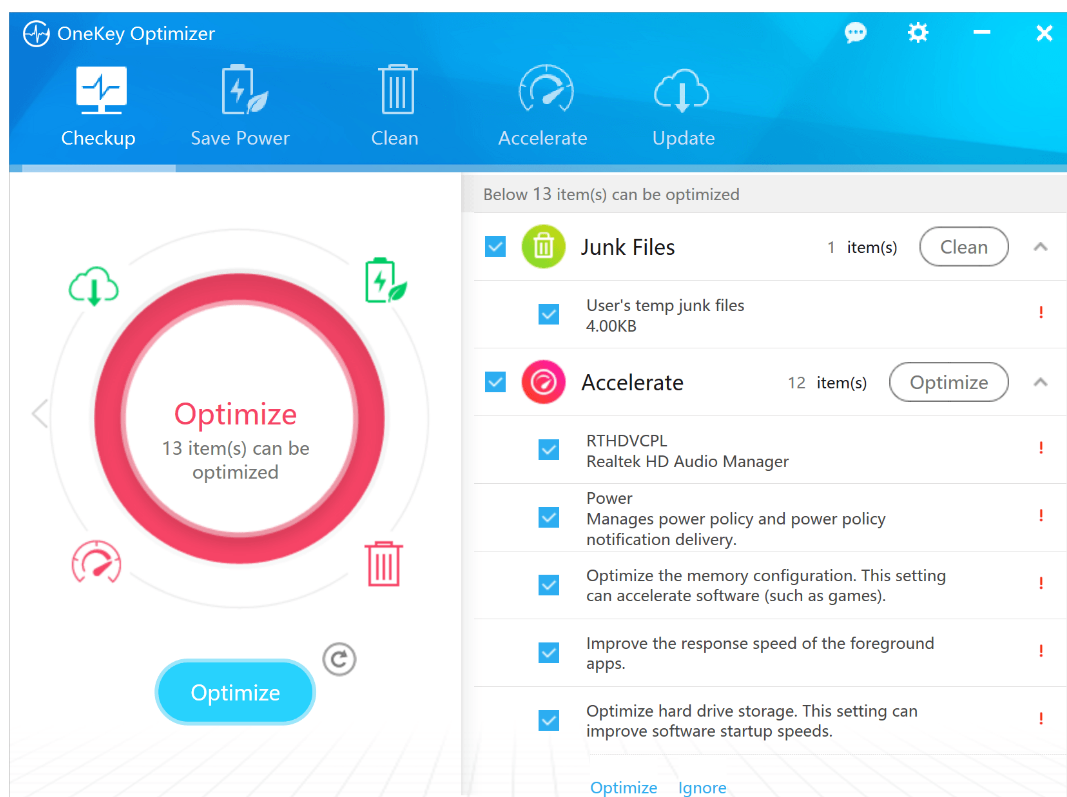

Figure 16 // Lenovo OneKey Optimizer screenshot from thinkwiki.org

6 https://support.lenovo.com/au/en/downloads/ds103119
7 https://support.lenovo.com/bo/sv/product_security/lse_bios_notebook

In 2018, an *article* from Tech Power Up revealed that ASUS was also installing software from a component located in the firmware. The software installed is called ASUS Armoury Crate. Details about this software can be found *here*.

In the course of our research, we've come across three other components installed from the firmware: Samsung's Secure Guard, HP Sure Run agent, and Phoenix Failsafe. We discuss two of them in the section dedicated to the technical analysis.

Observed techniques

When Computrace was first created, no dedicated Windows mechanism existed for the firmware to tell Windows it wanted to install software. Thus, developers needed to find alternative ways to achieve this goal. The technique that was used back then was to replace a Windows executable that is launched at startup. To do so, the firmware needs to implement its own NTFS driver to get file-based access to the Windows partition and overwrite the targeted executable. We've seen two system executables targeted.

The first one is `autochk.exe`. Autochk is launched early during the Windows boot process and performs integrity checks on file systems. It is the most common scenario we observed. That's the executable that Computrace/LoJack replaced for years. This is also the executable that was replaced by LoJax, a *UEFI rootkit* discovered by ESET researchers in 2018. The rootkit mimicked the architecture of LoJack to achieve persistence.

The other executable that is targeted is `spoolsv.exe`. Spoolsv is a component of the *print spooler subsystem*. It was used to install Phoenix Failsafe. Failsafe is anti-theft software that was *acquired* by Absolute Software Corporation back in 2010. It was also used by Samsung's Secure Guard, which we cover in this paper.

For the sake of completeness, we should mention that Hacking Team's UEFI rootkit achieved persistence by installing malicious OS-level components in the Start Menu folder.

Microsoft noticed that these techniques were used and introduced a mechanism that allows the firmware to notify the OS that it wants to install and launch an executable. This mechanism is called the Windows Platform Binary Table (WPBT). The firmware needs to install a custom Advanced Configuration and Power Interface (ACPI) table of type WPBT that points to a Portable Executable binary loaded in physical memory. During the OS initialization, the session manager will write the binary to disk in `\Windows\System32\Wpbbin.exe` and run it. The binary must be digitally signed. Microsoft published the documentation for the WPBT that can be found *here*. With the introduction of this feature, OEMs have shifted to using this mechanism when supported.

Case study: Lenovo Service Engine

The first case study that we cover is software called Lenovo Service Engine (LSE) that used to have a persistence module embedded in the firmware. In 2015, following the disclosure of vulnerabilities by security researchers, Lenovo issued a firmware update where they removed LSE. While LSE's firmware component uses the WPBT to install the OS-level software, it also uses some peculiar techniques to convey additional information from the firmware to the operating system. Since this way of doing things really stood out compared to other similar solutions, we decided to include a technical analysis of LSE in this paper.

The firmware component that attracted our attention is called LSEInit.efi. This UEFI executable contains the string `WINDOWS\SYSTEM32\LSEF.exe`. LSEInit only checks if this file exists on Windows partition, which we thought was interesting because LSEInit is a firmware component. Digging a little bit deeper, we found a second UEFI executable called LUFTSys.efi. This one also references a lot of Windows-related data such as PE executables and registry keys. All of this data is added in a custom ACPI table called LUFT. Interestingly, the string 'LSEF' was also added to that table. Figure 17 shows part of the code that initializes the 354786-byte long ACPI table. We knew we were on the right path, but yet had no clue as of how this LSEF.exe file was written to the file system.

```
v37 = AllocWideStr(L"\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\Session Manager");
*&LuftAcpiTable->field_3B2 = *v37;
*&LuftAcpiTable->field_3B6 = v37[1];
*&LuftAcpiTable->field_3BA = 1;
*&LuftAcpiTable->field_3BE = 982;
*&LuftAcpiTable->field_3C2 = 0x3EE;
*&LuftAcpiTable->field_3C6 = 0x406;
*&LuftAcpiTable->field_3CA = 0x41E;
*&LuftAcpiTable->field_3CE = 0x436;
*&LuftAcpiTable->field_3D2 = 0x44E;
v38 = AllocWideStr(L"BootExecute");
*&LuftAcpiTable->field_3D6 = *v38;
*&LuftAcpiTable->field_3DA = v38[1];
*&LuftAcpiTable->field_3DE = 7;
v39 = AllocWideStr(L"LSEF");
*&LuftAcpiTable->field_3E2 = *v39;
*&LuftAcpiTable->field_3E6 = v39[1];
*&LuftAcpiTable->field_3EA = 3;
*&LuftAcpiTable->field_682 = 'TFUL';
LOBYTE(v40) = 0x20;
*&LuftAcpiTable->field_686 = 0;
*&LuftAcpiTable->field_68A = 0;
*&LuftAcpiTable->field_83A = 0x55AA55AA;
*&LuftAcpiTable->field_9A6 = 'TFUL';
*&LuftAcpiTable->field_9AA = 0;
*&LuftAcpiTable->field_9AE = 0;
*&LuftAcpiTable->field_B5E = 0x55AA55AA;
```

Figure 17 // Hex-Rays output of part of the LUFT ACPI table initialization

That's where the final piece of the firmware-side of the puzzle comes to play. Another UEFI executable that we'll refer to as the LSE dropper, identified by the GUID 31e5caf3-a471-4e73-9f93-6f59dd4424f1, is responsible for adding a WPBT ACPI table. Since it is a UEFI application and not a DXE Driver, it is not loaded automatically by the DXE dispatcher. It needs to be executed manually by a firmware component. Interestingly, the dropper is read from the firmware and executed by the Computrace module.

The LSE dropper also decompresses a native application embedded in its text section. It then creates a WPBT ACPI table pointing to the application in physical memory. This is the final component of the firmware side of LSE. The firmware side of the architecture is shown in Figure 18.

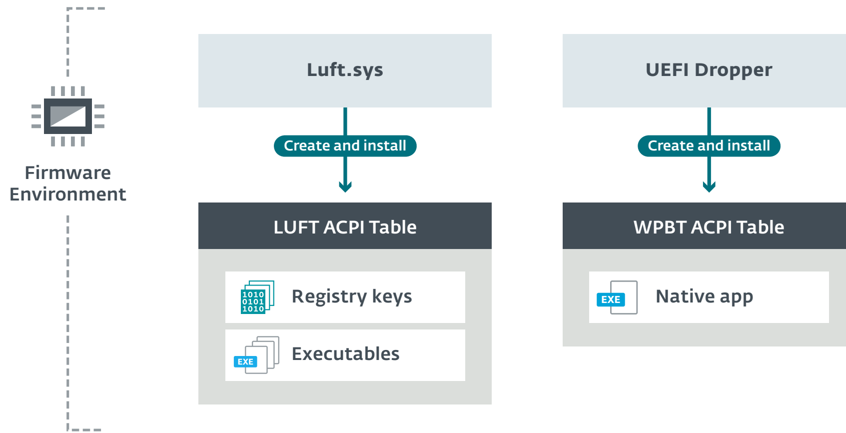

Figure 18 // Architecture of Lenovo Service Engine (LSE): Firmware side

OS Environment

Windows Session manager

Read

WPBT ACPI Table

Native app

Write to disk and execute

EXE

Native App

Read

LUFT ACPI Table

Registry keys

Executables

Extract and write to disk

Install

Install

EXE

rpcnetp.exe

[...]

UsbFilter service

[...]

EXE

LSEF.exe

Write to disk and create service

Write to disk

EXE

LSEDT.exe

Check signature and execute

EXE

LSEPreDownloader.exe

Query archive over HTTP

Send archive

Lenovo Server

Extract archive, validate the binary signature and execute it

Figure 18 // Architecture of Lenovo Service Engine (LSE): Firmware side

Let's now look at what happens on the OS side. Figure 19 illustrates the execution flow as well as the components involved. Since a WPBT ACPI table exists, the Windows session manager writes the native application referenced in the table to `\Windows\System32\Wpbbin.exe` and executes it. This application decompresses rpcnetp.exe, which is embedded in its text section, and writes it to disk. Then, it retrieves the LUFT ACPI table and parses it. Figure 20 shows the code responsible for reading and extracting data from this custom ACPI table, which contains all the configuration and the PE executables that LSE installs.

LSE starts by installing a USB filter kernel driver. To support both versions of Windows, a 32-bit and a 64-bit version of the driver are in the LUFT table. Then, LSEF.exe is written into `WINDOWS\SYSTEM32\` and persistence is added by modifying the `BootExecute` Registry key in `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager`

LSEF.exe embeds two other binaries (LSEDT.exe and LSEPreDownloader.exe), which are written to the System32 directory, and it installs LSEDT.exe as a Windows service. LSEDT.exe then runs LSEPreDownloader.exe, which queries Lenovo's server to get an additional executable that, once fetched, is written to disk and executed.

The communication is done over HTTP, allowing for a man-in-the-middle (MitM) attack. However, the signature of the binary is checked before execution. Figure 21 shows the code responsible for asserting that the signer certificate is issued by a trusted certificate. Additionally, it checks whether the Common Name (CN) or the Organization (O) field equals "Lenovo.Ltd". These security measures strongly mitigate the risk of an attacker delivering a malicious executable while spoofing the HTTP response.

```
fnPrint(L"\n Get LUFT table data start");
v0 = fnGetSystemFirmwareTable('ACPI', 'TFUL', 0, 0);
TableSize = v0;
if ( v0 )
{
  LuftTableBuffer = HeapAlloc_(TableSize);
  if ( LuftTableBuffer && fnGetSystemFirmwareTable('ACPI', 'TFUL', LuftTableBuffer, TableSize) )
  {
    gLuftTable = LuftTableBuffer;
    fnPrint(L"\n Get ACPI table point \n");
    ComputraceEnableFlagPtr = &gLuftTable->ComputraceEnableFlag;
    ComputraceEnableFlag = gLuftTable->ComputraceEnableFlag;
    LseFlag = gLuftTable->LseFlag;
    v27 = gLuftTable->field_26;
    DebugPrint = gLuftTable->DebugPrint;
    NbSections = gLuftTable->NbSections;
    OffsetTable = &gLuftTable->OffsetSectionUsbf;
    for ( SectionIndex = 0; SectionIndex < NbSections; ++SectionIndex )
    {
      Offset = OffsetTable[SectionIndex];
      Section = (gLuftTable + Offset);
```

Figure 20 // Hex-Rays output of the routine parsing the LUFT ACPI table

```
if ( CryptMsgGetParam(phMsg, CMSG_SIGNER_INFO_PARAM, 0, 0, &pcbData) )
{
  SignerInfo = (CMSG_SIGNER_INFO *)LocalAlloc(0x40u, pcbData);
  if ( SignerInfo )
  {
    if ( CryptMsgGetParam(phMsg, CMSG_SIGNER_INFO_PARAM, 0, SignerInfo, &pcbData) )
    {
      Issuer = SignerInfo->Issuer;
      SerialNumber = SignerInfo->SerialNumber;
      CertContext = CertFindCertificateInStore(
                      phCertStore,
                      PKCS_7_ASN_ENCODING|X509_ASN_ENCODING,
                      0,
                      CERT_FIND_SUBJECT_CERT,
                      &pvFindPara,
                      0);
      CertContext_1 = CertContext;
      if ( CertContext )
        CertSubjectName = CertGetNameString(CertContext);
    }
    else
    {
      CertContext_1 = 0;
    }
    LocalFree(SignerInfo);
    if ( CertContext_1 )
      CertFreeCertificateContext(CertContext_1);
  }
}
```

Figure 21 // Hex-Rays output of the signer's certificate issuer validation

Case study: Samsung's SecureGuard

The second case we cover in this paper is Samsung's SecureGuard. SecureGuard is a good example of how software installed by UEFI firmware can introduce an additional attack surface. SecureGuard is a solution installed by the firmware that is responsible for downloading and installing SW Update Guide, a utility tool to keep software up to date. The main issue is that one of the Windows executables installed by the firmware downloads and executes a file fetched over HTTP without validating it. Hence, an attacker in a MitM situation can send an arbitrary executable to the Samsung device. The vulnerable executable is embedded in the firmware image, thus making it difficult to issue a security fix.

Figure 22 shows the architecture of SecureGuard from the firmware components up to the final vulnerable executable running on Windows.

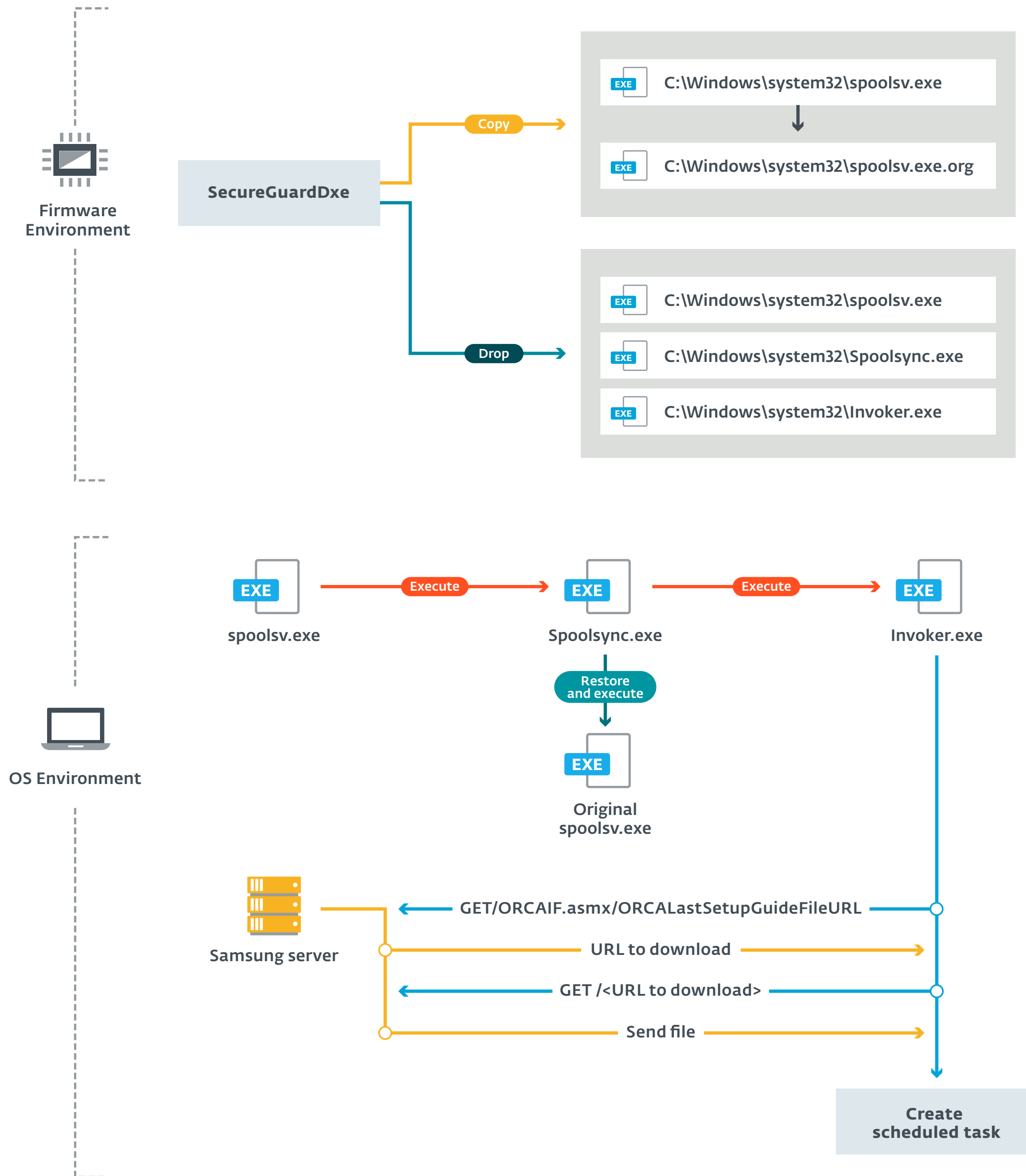

Figure 22 // Architecture of SecureGuard

SecureGuard has a component in the UEFI firmware. The UEFI executable is either called SecureGuardDxe or PhxSgDxe. This executable runs during the DXE phase of the platform initialization. Its role is to load three PE binaries from the SPI flash into memory and write them to the Windows partition. To run during early Windows boot, SecureGuard backups the original spoolsv.exe file and replaces it with its own file.

```
if ( FileExists(EfiSimpleFileSystemProtocol, L"\\Windows\\system32\\spoolsv.exe.org") < 0 )
  CopyFile(
    EfiSimpleFileSystemProtocol,
    L"\\Windows\\system32\\spoolsv.exe",
    L"\\Windows\\system32\\spoolsv.exe.org");
WriteFileToDisk(EfiSimpleFileSystemProtocol, L"\\Windows\\system32\\spoolsv.exe", &FileGuid1);
WriteFileToDisk(EfiSimpleFileSystemProtocol, L"\\Windows\\Spoolsync.exe", &FileGuid2);
WriteFileToDisk(EfiSimpleFileSystemProtocol, L"\\Windows\\Invoker.exe", &FileGuid3);
```

Figure 23 // Hex-Rays output of SecureGuard's UEFI executable code responsible for writing files to disk

When Windows starts its services, SecureGuard's version of spoolsv.exe runs and creates a new process by running spoolsync.exe. Spoolsync restores the original spoolsv.exe and then launches Invoker.exe impersonating the current user using the `CreateProcessAsUser` API call. Thus, Invoker.exe doesn't run as `NT AUTHORITY\SYSTEM`, but rather with the privileges of the current user.

Invoker.exe is the vulnerable component in this solution. It queries Samsung's server to retrieve a URL pointing to the executable to download. Figure 24 shows the first request that is performed over HTTP. Let's note that the "%20HTTP/1.1" is a mistake from the developers, who appended the HTTP string in the URL.

```
GET /ORCAIF.asmx/ORCALastSetupGuideFileURL?XMLVERSION=string%20HTTP/1.1
HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; WOW64;
Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR
3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
Host: orcaservice.samsungmobile.com
Connection: Keep-Alive
```

Figure 24 // First HTTP request to Samsung's server

The response to this request is an XML document containing the URL of the executable as shown in Figure 25.

```
<?xml version="1.0" encoding="utf-8"?>
<LastFileURLResp xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://
orcaservice.samsungmobile.com/">
  <SWINFO>
    <CISREV>BASW-13498A09</CISREV>
    <FURL>http://orcaservice.samsungmobile.com/dl/client/SecSWMgrGuide.exe_</
FURL>
  </SWINFO>
</LastFileURLResp>
```

Figure 25 // XML response containing the URL to download

Invoker.exe then downloads the given URL and creates a scheduled task so that it runs every time a user is logged in. Since all the communications are done over HTTP, an attacker in a MitM position could easily modify or spoof the response pointing to a malicious binary that would be executed instead.

What makes this attack possible is the fact that the downloaded binary is not subject to any type of validation. While it is common for such software to download a URL over HTTP (we've seen this above with Lenovo Service Engine), it is unusual not to perform any integrity check on the binary before executing it.

We looked at the latest versions of the UEFI firmware for devices with SecureGuard. Samsung removed this solution in the latest firmware updates. Since Samsung uses an internal platform ID number to identify their devices before downloading the proper firmware update, we were unable to find what Samsung devices have SecureGuard installed. Hence, we strongly suggest that you download the latest firmware update for your Samsung devices.

Case study: HP Sure Run

The last case we cover is an HP agent that hasn't been publicly analyzed before. HP provides a PDF document detailing the userland agent, but the fact that it is installed from the firmware wasn't covered.

HP Sure Run agent has a firmware module that is responsible for installing an ACPI WPBT table pointing to a native application embedded in the firmware image. HP's usage of the WPBT mechanism is more aggressive than what we're used to seeing with other OEMs. Instead of solely installing the table, the firmware component registers a callback on ACPI table creation to stop other firmware components from installing a WPBT. Figure 26 shows the code of this callback. Furthermore, if a WPBT is already installed when the HP firmware component wants to create its own, it retrieves the existing table and replaces the binary pointed to by HP's native application.

```
__int64 __fastcall AcpiTableCreationCallback(_DWORD *AcpiTable)
{
  EFI_ACPI_TABLE_PROTOCOL *AcpiTableProtocolGuid; // [rsp+30h] [rbp+8h]

  AcpiTableProtocolGuid = 0i64;
  if ( AcpiTable )
  {
    if ( *AcpiTable == 'TBPW' )
    {
      InstallHpSureRunAgentTable();
      gHpTableNotInstalled = 0;
      if ( !gBootServices->LocateProtocol(&gEfiAcpiTableProtocolGuid, 0i64, &AcpiTableProtocolGuid) )
        (AcpiTableProtocolGuid->UninstallAcpiTable)(AcpiTableProtocolGuid, gAcpiTable);
    }
  }
  return 0i64;
}
```

Figure 26 // Hex-Rays output of the callback blocking WPBT creation

The native application installed by the firmware creates a Windows service as shown in Figure 27. The service points to HP_SureRun.exe, a .NET executable embedded in its resource section. The documentation provided by HP accessible *here* describes HP Sure Run as a "hardware-enforced application persistence solution". We did not audit the solution for vulnerabilities.

```
v7 = CreateService(
       L"HPSureRun",
       L"HP Sure Run Service",
       L"%SystemRoot%\\SysWOW64\\HP_SureRun.exe",
       L"Provides Sure Run security features for selected services and applications");
```

Figure 27 // Hex-Rays output of the service creation function call

Security implications

In the final part of this paper, we discuss the security issues that arise from the inclusion of UEFI firmware backdoors and persistence modules in the firmware.

Let's first look at the UEFI firmware backdoors. One of the main problems with them is that they allow an attacker to unlock the UEFI firmware setup where key security mechanisms are configured. SecureBoot, for instance, is one of the most important mechanisms in the boot chain of trust. If an attacker gets physical access to a machine with a UEFI firmware backdoor, he can disable SecureBoot and replace the bootloader with a malicious one.

While it could be argued that SecureBoot is not a security mechanism against physical access, other security mechanisms specifically tailored against physical access are also configured via the UEFI firmware setup. For instance, it is possible to lock the device boot order and disable USB devices in the pre-OS environment from the UEFI firmware setup. Configuring both these settings is meant to reduce the risk of an attacker booting from an alternative source than the bootloader already installed on the machine. While the efficiency of such a mechanism against physical attack can be debated, it is clear that UEFI firmware backdoors allow bypassing it completely. The same can be said of other features such as fingerprint authentication and chassis intrusion detection.

Another side of it is the false sense of security it gives to the user. By configuring a password to prevent the system from booting, users may think their computers are unbootable by anyone who doesn't possess the password. Because of UEFI firmware backdoors, this protection mechanism does not fulfill its duty. We believe it is important that users know the limits of the protections they use and hope this paper helps in that regard.

As for the persistence modules embedded in the firmware, the main problem resides in the fact that the delivery of firmware updates is fairly complicated. Most of computer users do not update their firmware, while the only way to fix vulnerable software stored in the firmware is to do so. This means that a computer shipped with a vulnerable firmware component will most likely remain vulnerable during the system's complete lifetime.

LSE is a good example of such vulnerable software stored in the SPI flash memory. After security researchers disclosed vulnerabilities in this solution, Lenovo issued a firmware update in which LSE was removed. To patch their systems, users had to follow the steps mentioned in this *security advisory*. This is not an efficient mechanism to massively distribute a security fix and it illustrates how complicated delivering firmware security updates is. For this reason, we believe firmware persistence should be avoided as much as possible and limited to cases where it is strictly necessary as is the case with anti-theft solutions.

CONCLUSION

While our UEFI executable processing pipeline did not allow us to find UEFI malware yet, the results it has produced so far are promising. Some of the identified outliers used techniques that could've been used by malware to achieve persistence at the OS level. The same heuristics that allowed us to catch Samsung SecureGuard would've caught Hacking Team and Sednit's UEFI rootkits, which are both using an NTFS driver to deploy malware on Windows partition. Additionally, the similarity score computed on LoJax and public proof-of-concept UEFI malware were very low, assertively classifying them as outliers. The future will tell if this story has another chapter.

Special thanks to Hamidreza Ebtehaj and Martin Smolár for their help in the analysis.

ABOUT ESET

For 30 years, *ESET®* has been developing industry-leading IT security software and services for businesses and consumers worldwide. With solutions ranging from endpoint and mobile security, to encryption and two-factor authentication, ESET's high-performing, easy-to-use products give consumers and businesses the peace of mind to enjoy the full potential of their technology. ESET unobtrusively protects and monitors 24/7, updating defenses in real time to keep users safe and businesses running without interruption. Evolving threats require an evolving IT security company. Backed by R&D centers worldwide, ESET becomes the first IT security company to earn *100 Virus Bulletin VB100* awards, identifying every single "in-the-wild" malware without interruption since 2003. For more information, visit www.eset.com or follow us on *LinkedIn*, *Facebook* and *Twitter*.

ENJOY SAFER TECHNOLOGY™